The Exact Solution of Linear Equation Systems on a Shared Memory Multiprocessor*

Wolfgang Schreiner
schreine@risc.uni-linz.ac.at

Volker Stahl
vstahl@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria
Tel. +43 7236 3231 66; Fax: +43 7236 3231 30

June 9, 1995

Abstract

We describe the design of a parallel algorithm for the exact solution of linear equation systems with integer coefficients and the implementation of this algorithm on a shared memory multiprocessor. An efficient solution of the original problem is difficult since the coefficients grow during the computation and arithmetic becomes very time-consuming. Therefore we transform the problem into a problem of determinant computation and apply a modular approach: the system is mapped into several finite fields where the determinants can be efficiently computed. The subresults are combined to yield the original determinants and to compute the solutions of the system. Several parallel versions of this algorithm have been developed and implemented on a shared memory multiprocessor. The programs are applied to equation systems of different characteristics and the results are analyzed and compared.

Keywords: Parallel algorithms, scientific computing, computer algebra, shared memory machines.

*Funded by the Austrian Research Agency FWF in the frame of the project S5302-PHY "Parallel Symbolic Computation".

1 Introduction

In scientific computing, much research has been devoted to the parallelization of numerical algorithms. Since these methods operate with efficient finite-precision (floating point) arithmetic, they yield fast approximative solutions of scientific problems. However, sometimes it is also important to determine the *exact* solutions of such problems. This is in particular the case if numerical methods, by the accumulation of approximation errors, may yield qualitatively wrong answers.

Computer algebra is that branch of computer science that aims to provide exact solutions of scientific problems. Research results of this area are e.g. algorithms for symbolic integration, polynomial factorization or the exact solution of algebraic equations and inequalities [2]. All these algorithms have to operate with arbitrary precision arithmetic; they are therefore much more expensive with respect to time and space than the corresponding numerical methods.

Thus the parallel computation group at RISC-Linz has started a project that pursues the development of PACLIB, a library of parallel algorithms based on the computer algebra library SACLIB [4]. PACLIB has been implemented in the C language on top of a runtime kernel that supports an efficient and high-level parallel programming model [8]. The PACLIB kernel [9] was implemented on a 20 processor Sequent Symmetry, a MIMD computer with shared memory, but is in principle portable to any UNIX machine.

In this paper we discuss the PACLIB approach to one of the basic problems in computer algebra: the exact solution of systems of linear equations with arbitrarily large integer coefficients. The problem itself is not of particular interest; however, its solution is the main constituent of generalized interpolation algorithms that are used for the formal manipulation of multi-variate polynomials (e.g. for factorization and resultant computation). Handling such polynomials is of extreme importance in several practical applications, e.g. in geometric modelling [7] or in the stability analysis of control systems [6].

In this context, it is unsafe to approximate the solutions by numerical methods [5], since the problems are highly *non-linear*: Numerical errors, even if they are initially extremely small, grow very much during the computation and may e.g. let all the original roots of a polynomial disappear [11]. Hence, we require methods for finding the exact solutions of linear equation systems.

Currently, the (both asymptotically and practically) most efficient sequential algorithm for this problem is based on a modular method and a combination of Cramer's Rule with Gaussian elimination [12]. However, the algorithm is still very time-consuming and even with fast computers only small systems can be handled. Extending the application range of the method is therefore of extreme importance for scientific computation.

We have analyzed the problem and created several PACLIB versions of the modular method. This paper describes our results in the following fashion: After a short discussion of the modular method, the parallel programming model of PACLIB is sketched. Then we gradually develop several parallel versions of the algorithm and compare the resulting PACLIB programs with respect to their dynamic behavior. Runtime benchmarks are performed on equation systems with different characteristics and the achieved results are discussed.

2 The Modular Method

The problem we consider is the following: Let A be a regular $n * n$ matrix over $\mathbb{Z}$ and b a vector of length n over $\mathbb{Z}$. We want to find that vector x of length n over $\mathbb{Q}$ such that

$$A * x = b$$

Such an x exists and is unique since we assume that A is regular.

In other words, we want to solve a system of n linear equations with n variables where the coefficients of the equations are denoted by A and the right hand sides of the equations by b, respectively. Please note that

1. A and b contain arbitrary integer numbers whose sizes are not limited by the word length of any computer.
2. We want to find a result vector x of rational numbers that are the *exact* solutions of the given equation system.

Since the input numbers are arbitrary elements of $\mathbb{Z}$, they have to be represented by sequences of computer words. Arithmetic operations can therefore not be performed in constant time but their complexity depends on the *length* of the involved integers, i.e. the number of computer words required for their representation[1]. Furthermore, x is a vector over $\mathbb{Q}$ where each rational number is represented by two relatively prime integers (the minimum nominator and denominator, respectively).

The elements of A and b may be very long, in particular if they are derived from an equation system with rational coefficients by multiplying each equation with the least common multiple of all denominators. Let us assume that l is a bound for the word length of all coefficients. If we apply the standard *Gaussian elimination* algorithm for solving the equation system, the involved integers will grow to length $i * l$ after the elimination of the i-th row, i.e. arithmetic will become more and more complex. The complexity[2] of the Gaussian elimination algorithm is then bounded by $O(n^5 l^2)$ [12].

The complexity can be improved by applying *Cramer's Theorem*. This theorem tells us that each solution x_i can be computed by

$$x_i = \frac{y_i}{d}$$

where $d = \det(A)$ and $y_i = \det(A_i)$ and A_i is A with the i-th column replaced by b. Usually, the algorithm derived from this rule is less efficient than Gaussian elimination. However, in our case this method is of advantage since we can avoid the "blowup" of intermediate results by using *modular arithmetic* and the *Chinese remaindering algorithm*:

1. **Mapping:** We take k prime numbers p_j and map the given system (A, b) over $\mathbb{Z}$ into k systems (A^j, b^j) over the finite fields $\mathbb{Z}/p_j$. The mapping is performed by

---

[1] For example, multiplication requires $O(l_1 l_2)$ steps where l_1 and l_2 are the operator lengths, respectively.

[2] The well-known complexity bound $O(n^3)$ only holds if we assume arithmetic to take constant time.

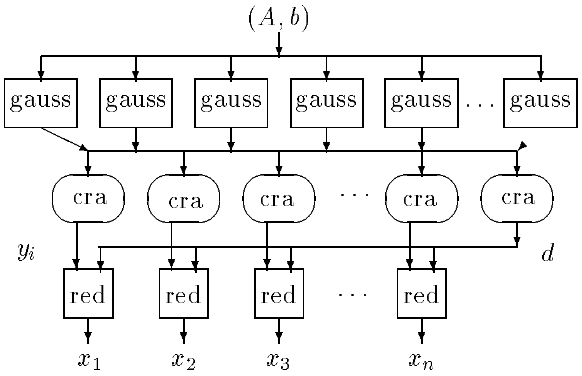

Figure 1: The Modular Method

computing the remainder of each integer with respect to p_j i.e. all elements of $\mathbb{Z}/p_j$ are in $\{0 \dots p_j - 1\}$. If the p_i fit into single computer words, arithmetic can in these fields be performed in constant time.

2. **Gaussian Elimination:** In each image (A^j, b^j), we compute the determinant $d^j = \det(A^j)$ and all determinants $y_i^j = \det(A_i^j)$. The later is done by applying Gaussian elimination and Cramer's theorem in the "reverse" direction i.e. by solving the system $A^j * x^j = b^j$ and then computing $y_i^j = d^j * x_i^j$.

3. **Chinese Remaindering:** Applying the Chinese Remaindering Algorithm, d is reconstructed from the k values d^j and each y_i is reconstructed from the k values y_i^j. This is possible since the (A^j, b^j) are *homomorphic images* of (A, b) with respect to determinant computation.

4. **Reduction:** The solutions x_i are computed as y_i/d_i and reduced by the Euclidean Algorithm for the computation of the greatest common divisor.

Due to lack of space, we cannot describe the Chinese remaindering algorithm in more detail (see e.g. [1]). Actually, the underlying Chinese remaindering theorem only tells us that the results x_i are unique images of the solutions in the field $\mathbb{Z}/p$ where $p = \Pi p_j$. However, if we use sufficiently many primes (such that p becomes greater than all solutions), the x_i are the actual solutions of the given system. It thus suffices to give a bound q for $|\det(A)|$ and $|\det(A_i)|$ and use so many primes that $p > 2q$. Such a bound is for example $q = n! * l$. The number k of required primes is in this case $O(nl)$ and the complexity of the whole algorithm is $O(n^3l^2 + n^4l)$ [12].

A minor technical difficulty is the fact that for some "unlucky" primes p_j the homomorphic image (A_j, b_j) may not be regular; d^j is then 0 and x_j cannot be computed by Gaussian elimination. Fortunately, this may happen only for finitely many p_j (and in practice never happens if the p_j are sufficiently large). If the unlucky case should really occur, the unlucky prime can be skipped and computation can proceed with the next prime.

The sequential implementation of this algorithm is usually structured as shown in Figure 2: The function `linsolve` assumes that there exists a precomputed list of single word prime

```
linsolve(A, b, x)
  plist := single word prime list
  q := bound(A, b)
  d,y,p := 0,<0..0>,1
  while p < 2*q do
    pj,plist := next(plist)
    Aj,bj := image(A, b, pj)
    dj,yj := gauss(Aj, Bj)
    if dj != 0 then
      d = cra2(p, d, pj, dj)
      y = cra2(p, y, pj, yj)
      p := p*pj;
  x := reduce(y, d)
```

Figure 2: Sequential Algorithm

numbers that can be used for each application of the program. After determining the bound q, `linsolve` maps the input system into the finite field over `pj` and computes `dj` and `yj` in this field. If `dj` is not `0`, a binary version of the Chinese Remaindering Algorithm `cra2` is applied to incorporate the new images `dj` and `yj` into the current approximations `d` and `y`. When sufficiently many images have been processed, the solution vector `x` is computed by reducing `y` over `d`.

Using the modular approach, we get one of the fastest algorithms for the exact solution of linear equation systems. In order to demonstrate the computational demands of this algorithm, we performed some timings using the SACLIB implementation of `linsolve` applied to random inputs. The times were measured on an i80386 running at 20 MHz.

$n \backslash l$	1	2	4	8
5	203	583	1817	6246
10	1590	4409	13518	46471
20	16581	42041	126427	360512

On the horizontal axis, the maximum word length l of the input elements is shown, on the vertical axis the dimension n of the equation system. The table entries denote the runtimes in milliseconds. From these figures, the motivation to parallelize the algorithm becomes obvious. Doubling the size of the input integers triples the runtime; doubling the matrix dimension increases the runtime by a factor of 10. Therefore even with fast microprocessors only small equation systems can be solved in a acceptable amount of time.

3 PACLIB

PACLIB is a system for the parallel implementation of algorithms of computer algebra and related areas. It has been developed at our institute on the basis of two public domain software packages:

- SACLIB [4] is a library of C functions that is based on a heap management kernel with automatic garbage collection and supports all fundamental objects and methods of computer algebra (in particular polynomial arithmetic).
- The μSYSTEM [3] is a library of C functions that supports light-weight concurrency on shared memory multiprocessors and workstations running under UNIX.

The PACLIB programming model is basically a functional one: Each user function can be executed as a concurrent task; communication and synchronization is mainly based on the availability of task results. In the following, we give the (abstract) syntax of some of the most important PACLIB constructs. The function

task = `pacStart(`*fun*`, `*args*`)`

creates a new task executing *fun(args)* in parallel with the current task. The new task terminates and deallocates its resources as soon as the *fun* call returns with a result r. *task* is a reference to the new task that may be used by other tasks for retrieving r. For instance,

result = `pacWait(`*tptr*`, `*tasks*`)`

non-deterministically delivers the result r_i of one of the denoted *tasks*. The reference of the delivering task t_i is stored at the location *tptr* and r_i is returned as *result*. If all tasks are still running, `pacWait` temporarily blocks the current task. The function

`pacStop(`*tasks*`)`

prematurely aborts the execution of all denoted *tasks*. In combination with the non-deterministic `pacWait`, this function can be used to express *speculative parallelism*.

Tasks may also communicate via *streams* i.e. buffered (and potentially unbounded) communication channels. Exactly one writer task may use `pacPut` to put values into the stream while a set of reader tasks may use `pacGet` to retrieve these values from there. Streams are a functional concept similar to linked lists: each reader has the same view of the stream and will receive the same sequence of values independently of the other readers. *Buffer interfaces* allow a more general access to streams.

Due to lack of space, we cannot describe the PACLIB programming model in more detail. There are many variations of the above functions that facilitate the most frequent application forms. For a complete definition of all constructs and their concrete syntax, see [8]. PACLIB has been implemented on a Sequent Symmetry computer with 20 processors but is portable to most machines running under the UNIX operating system. The current implementation is based on shared memory but the programming model can be implemented on distributed memory architectures, too.

4 Parallelizing the Modular Method

The basic parallelization idea can be derived from Figure 1:

1. **Mapping:** The computation of the homomorphic images (A^j, b^j) can be performed independently in parallel.
2. **Gaussian Elimination:** The computation of the determinants d^j and y_i^j of the homomorphic images can be performed by concurrent tasks.
3. **Chinese Remaindering:** $n+1$ tasks may compute d and each y_i independently in parallel.
4. **Reduction:** The n reductions y_i/d can be performed in parallel.

For implementing the algorithm, the following facts should be especially considered:

- **Gauss:** For each prime p_j, the creation of the homomorphic image and the computation of the determinants d^j and y_i^j have to be performed in sequence and can therefore be combined to one `gauss` task.
- **CRA:** For the same reason, the computation of each y_i and the final reduction y_i/d can be performed by the same `crai` task. Each `crai` task must therefore eventually receive the result of the `crad` task that computes d.
- **Communication:** All `crai` tasks and the `crad` task need the results of all `gauss` tasks to compute y^j. However, these results need not be available in advance but it suffices if they eventually arrive in sequence.
- **Unlucky Primes:** We must care for the case that some p_j is "unlucky" and Gaussian elimination cannot be performed in the corresponding image. In this (very unlikely) case, another `gauss` task has to be started with a new prime.

The algorithm can then be naturally structured as shown in Figure 3:

1. **Bound:** First the prime bound $2 * q$ is computed by the main task.
2. **Gauss:** As many `gauss` tasks are started as required to exceed the prime bound and their handles are stored in `tlist`. Each task maps the given system into the finite field $\mathbb{Z}/p_j$ and computes the determinants d^j and y_i^j in these fields.
3. **CRAD:** A `crad` task is started with `tlist` as input argument. `crad` will receive the values d^j from the corresponding `gauss` tasks and compute d.
4. **CRAI:** n `crai` tasks are started with `tlist` and the handle of the `crad` task as input arguments. Each `crai` task will receive the values y_i^j from the `gauss` tasks, compute y_i and reduce this by the result d of the `crad` task.

```
plinsolve(A, b, x)
  plist := single word prime list
  p,tlist,slist,tstream := 1,nil,nil,pacOpen()
  q := bound(A, b)
  while p < 2*q do
    pj,plist := next(plist)
    t := pacStart(gauss, A, b, pj)
    tlist := comp(t, tlist)
    p := p*pj
  dt := pacStart(crad, tlist, tstream)
  for i:=1 to n do
    s = pacStart(crai, i, tlist, tstream, dt)
    slist = comp(s, slist)
  while pacWaitListRm(&r, &tlist) do
    if r.det = 0 then
      pj,plist := next(plist)
      t := pacStart(gauss, A, b, pj)
      tlist := comp(t, tlist)
      pacPut(tstream, t)
  for i:=1 to n do
    s,slist := next(slist)
    xi := pacWaitOne(s)
    x := comp(xi, x)
  return(x)
```

Figure 3: Parallel Algorithm

5. **Gauss:** The main task receives by `pacWaitListRm` the result r of any `gauss` task and removes the task handle from (its local copy of) `tlist`. If the result determinant is zero, the corresponding prime p_j was unlucky. In this case, a new `gauss` task is started and put into the `tstream`. After the `crad` and the `crai` tasks have processed the results of all tasks in `tlist`, they check `tstream` for additional tasks and incorporate the corresponding results.

6. **Result:** Finally, the main task receives the subresults of all `crai` tasks and stores them in the total result list.

Please note that we could also structure the algorithm in such a way that the main task waits for the results of all `gauss` tasks and checks for unlucky primes *before* the `crad` task and the `crai` tasks are started. All result values d^j and y_i^j could then be passed directly to `crad` and `crai` without using the indirections `tlist` and `tstream` (and thus without calling `pacWait` and `pacGet` within these tasks).

However, the drawback of such an approach outweighs the possible efficiency gains: the main task then has to wait until *all* `gauss` tasks have terminated before it can start any `cra` task for execution. Actually, the `cra` tasks do not need in advance the results of all

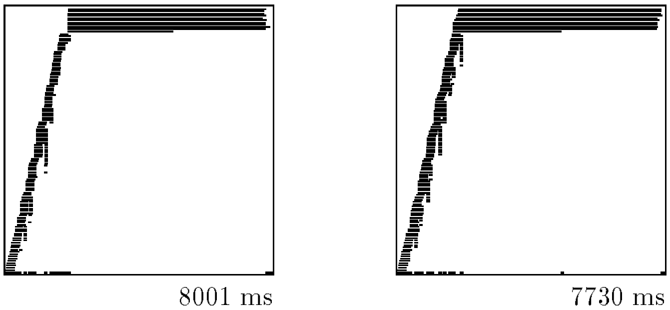

Figure 4: Synchronization vs Overlapping Execution

`gauss` tasks to start computation but they can proceed with execution if *any* `gauss` result is available and incorporate this result. Figure 4 demonstrates the difference between an implementation of the algorithm with synchronization after the `gauss` phase and an implementation that is modelled after the above skeleton.

The pictures were created with the program `pacgraph` that visualizes the profiling information generated by a PACLIB program. They consist of sets of horizontal lines each of which represents a task and denotes in the horizontal direction the time when the task was executed by some processor. The bottom line represents the main task, the many short lines the `gauss` tasks, the line of medium length the `crad` task and the very long lines the n `crai` tasks. From these pictures, we learn that the synchronized program is slightly more efficient in the Chinese remaindering phase but in overall looses due to the synchronization point.

5 Analysis of the Algorithm

We are now going to analyze the behavior of the algorithm with respect to different sorts of equation systems. Due to parallelization overhead, any parallel implementation of an algorithm looses on small examples against an optimized sequential version. In our case, it does not make sense to solve equation systems of small dimensions with small input coefficients by the parallel algorithm. Instead, we will concentrate on the following test configurations that cover typical application cases.

1. $\mathbf{n = 5, l = 30}$: This configuration represents the case of a small equation system with very large coefficients.
2. $\mathbf{n = 10, l = 10}$: This configuration represents the case of a medium-size equation system with large coefficients.
3. $\mathbf{n = 40, l = 1}$: This configuration represents the case of a large equation system with small coefficients.

The larger the dimension n and the larger the length l, the better the efficiency of the parallel algorithm becomes. The reason why we decrease in our test examples l for larger

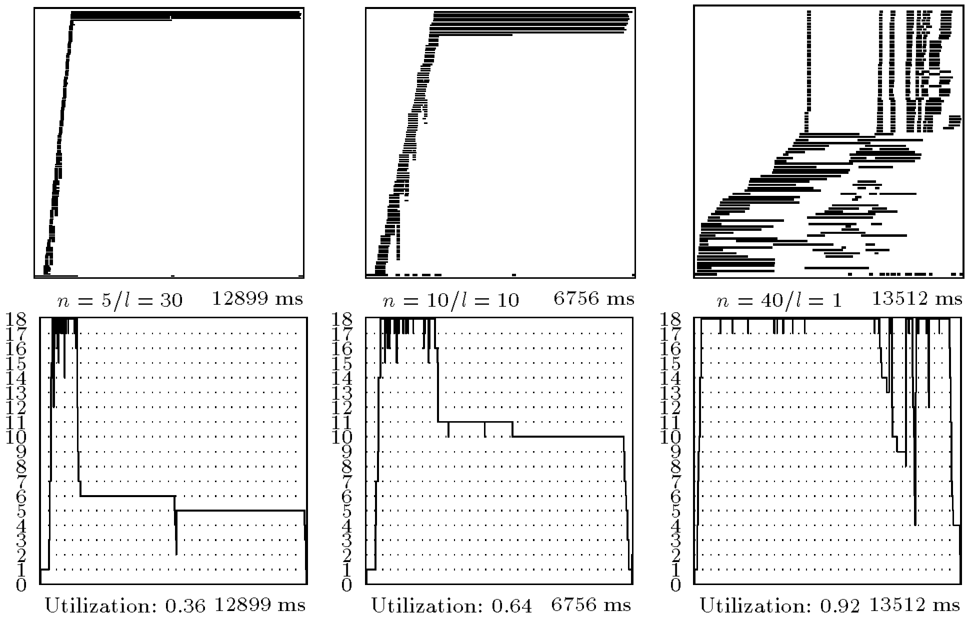

Figure 5: Basic Algorithm

n is the fact that the sequential `linsolve` otherwise would run extremely long making runtime comparisons difficult. The results displayed in Figure 5 show the following[3]:

1. **Large Dimensions:** If the dimension of the equation system n is larger than the number of processors p, the overall utilization of the processors is high and we can expect very good speedups (provided that the parallelization overhead is low).

2. **Small Dimensions:** If $n < p$, two problems arise:

   (a) **Granularity:** The runtime of each `gauss` task is $O(n^3)$ i.e. independent of the coefficient length l. This is no problem for large n, since then the task granularity is very large. For small n, the tasks run very short and the task creation overhead may decrease the efficiency of the program.

   (b) **Utilization:** During the Chinese remaindering phase, the machine is underutilized with $p - n - 1$ (finally $p - n$) unemployed processors. The reason for this is that there are only 1 `crad` and n `crai` tasks where the later also perform the final reduction. Since both Chinese remaindering and reduction take for large integers quite long, we cannot expect a speedup significantly larger than n.

The only problem with the case of large n is the possibility of excessive parallelism in the Gaussian Elimination phase: the number of `gauss` tasks equals the number of primes p_j required to exceed the prime bound (which is essentially $O(nl)$, see Section 2). Since

[3]The second kind of pictures was generated by the tool `pacutil` visualizing the processor utilization during the run of a PACLIB program.

each task requires a local workspace, the memory of the machine might thus be flooded. However, for "practical" problems, the number of gauss tasks is less than 1000 which is (with a workspace of 8 KB per task) not really a problem. Our further investigations will therefore concentrate on the optimization of the algorithm for equation systems with small dimensions but large coefficients.

6 Shifting Work

First we will investigate how to improve the granularity and thus the efficiency of the gauss phase of the program. We could achieve this, for instance, by combining m successive primes p_j into a set P_k and let each gauss task compute d^j and y^j for each $p_j \in P_k$. However, there are some technical difficulties with this approach e.g. it is not so easy to estimate the appropriate m before the product of primes has been computed.

Anyway, by a closer analysis of the program, we can shift a significant amount of work from the Chinese remaindering phase into the Gauss phase and thus improve the granularity of the gauss tasks: a main operation in Chinese remaindering is the computation of the inverse $P_j^{-1} \bmod p_j$ where P_j is the product of all primes $p_{j'}$ with $j' < j$ (i.e. of all primes processed up to that moment). The previous implementation was very wasteful by repeating this computation (that is independent of d^j and y^j) in each cra task.

A better approach is to let each gauss task already derive the corresponding inverse of P_j (which is passed as an additional argument by the main task) and deliver this inverse together with y^j, d^j and P_j to all cra tasks. This work shift has several advantages:

1. It decreases the total amount of work to do.
2. It decreases the runtime of the crad task and of the crai tasks.
3. It increases the granularity of the gauss tasks and makes it also dependent on the length l of the integer coefficients.

The results of this work shift are displayed in Figure 6. The runtime of the program is decreased by 30% in the case of small dimensions and large coefficients, by 20% in the medium case but not at all in the case of large dimensions with small coefficients. Actually, most of the improvement stems from the fact that the total amount of work was substantially decreased while the increased grain size of the gauss tasks plays a minor role.

We also experimented with several other versions of the algorithm that aimed to increase the granularity by blocking several gauss tasks into one and by introducing a manager-worker model [10]. However, these efforts did not result in any further performance improvements, mainly because the runtime of each task ($>$ 100 ms) already outweighs the small PACLIB parallelization overhead ($<$ 1 ms) [9]. The main performance limit is therefore still the long cra phase where processors may be underutilized.

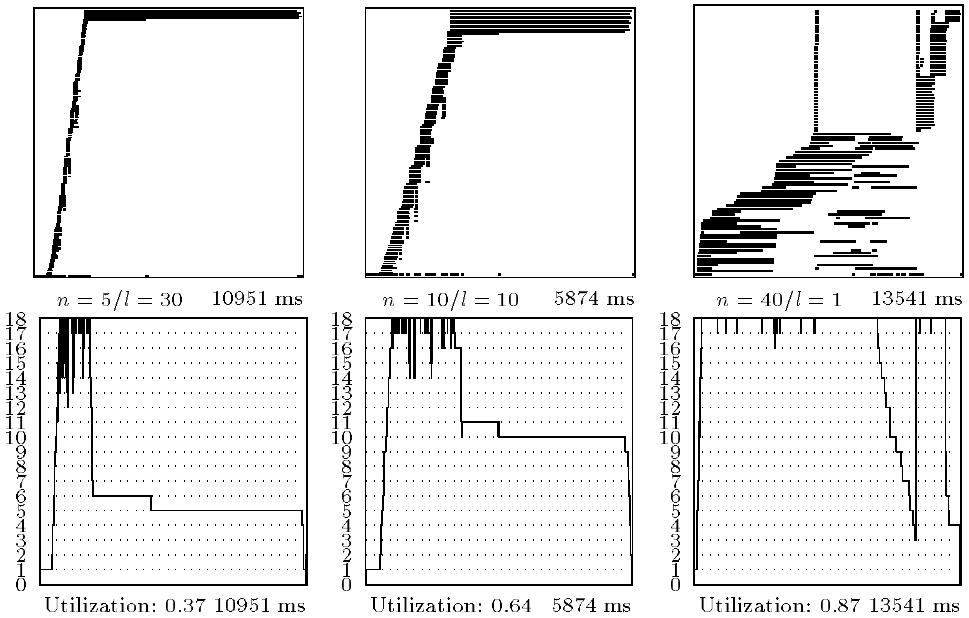

Figure 6: Shifting Work

7 Parallelizing Chinese Remaindering

Our main concern is now to decrease the runtime of the `cra` phase where the machine is potentially underutilized with work. Remember that there exist two kinds of `cra` tasks, one `crad` task computing d and the `crai` tasks computing y_i and performing the final reduction step y_i/d. The reduction phase is based on the Euclidean algorithm for the computation of the greatest common divisor; this is an essentially sequential algorithm for which we are not aware of any good parallelization.

Our efforts therefore can only concentrate on the parallelization of the Chinese remaindering algorithm itself. At the first glance, this seems very simple: the task of the algorithm is to combine a set of values V into a single value v (e.g. each `crai` task combines all y_i^j into a single y_i). This combination can also proceed in a divide and conquer fashion by splitting V into two subsets V_a and V_b that are combined into v_a and v_b respectively. Finally, v_a and v_b are combined into the total result v.

The problem with this approach is the fact that the length of the involved integers steadily increases during the combination. While all elements of V are rather small, the subresults v_a and v_b are extremely large (roughly their length is the sum of the lengths of all elements in V_a and V_b, respectively). The combination of v_a and v_b will therefore be very time-consuming and take almost half of the time of the total Chinese remaindering process!

A parallelization of the Chinese remaindering algorithm by the divide and conquer principle will therefore not yield good speedups. Anyway, since our goal is just to improve the processor utilization during the `cra` phase of the `plinsolve` program, we implemented a "flat" version of the divide and conquer approach: Each `cra` task splits the input set into

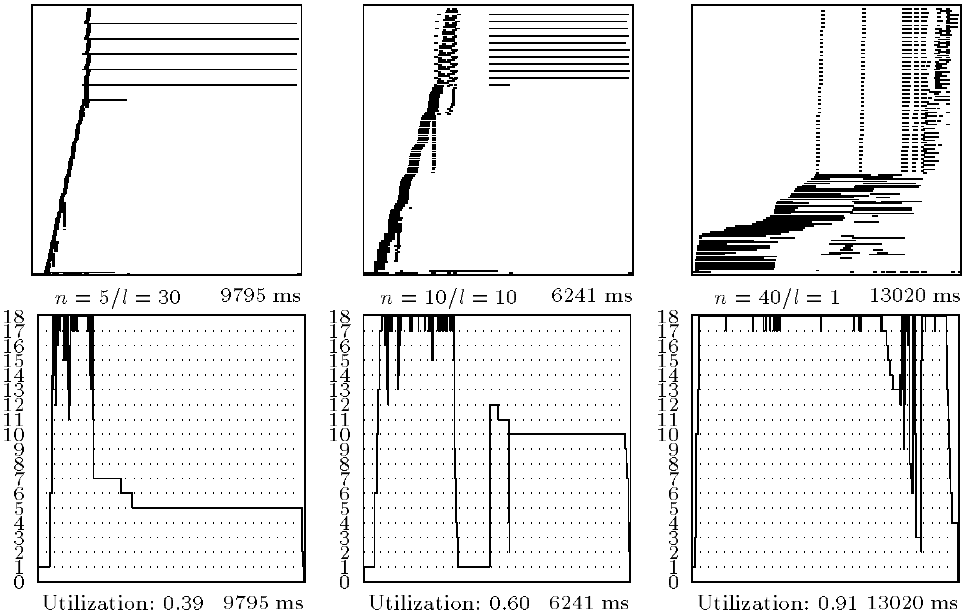

Figure 7: Parallelizing Chinese Remaindering

several subsets that are (sequentially) combined by separate tasks. The `cra` task then combines the results of these subtasks into the final result. The number of subtasks per `cra` task is $c * p/n$ where c is a small constant chosen to keep all processors busy.

By these measures, the total runtime can be somewhat reduced for the case of small dimensions (see Figure 7). Due to a better load balancing, the algorithm also slightly profits from the parallelization in the case of large dimensions. However, in the medium case an unexpected effect takes place due to the fact that this parallel version also applies the "work shift" explained in the previous section.

The `cra` tasks have to combine values that are themselves already combinations of the original values. Thus a main operation is the inverse computation $Q_j^{-1} \bmod P_j$ where P_j is the product of a subset of the primes p_j and Q_j is the product of all $P_{j'}$ with $j' < j$. This computation is not performed by a `gauss` task but by a separate `inv` task (represented by the second lowest line in the pictures). However, for the computation of the last inverse the `inv` task needs extremely long such that the `cra` tasks have to wait and processor utilization is low for some period.

8 An Alternative Algorithm

A further improvement can be achieved by extending the "work shift" idea to its extreme using a different formulation of the Chinese remaindering algorithm. The usual formulation (that was used up to now) yields an algorithm that iteratively includes the information derived from the next image v_j of the desired value into the information v gathered up

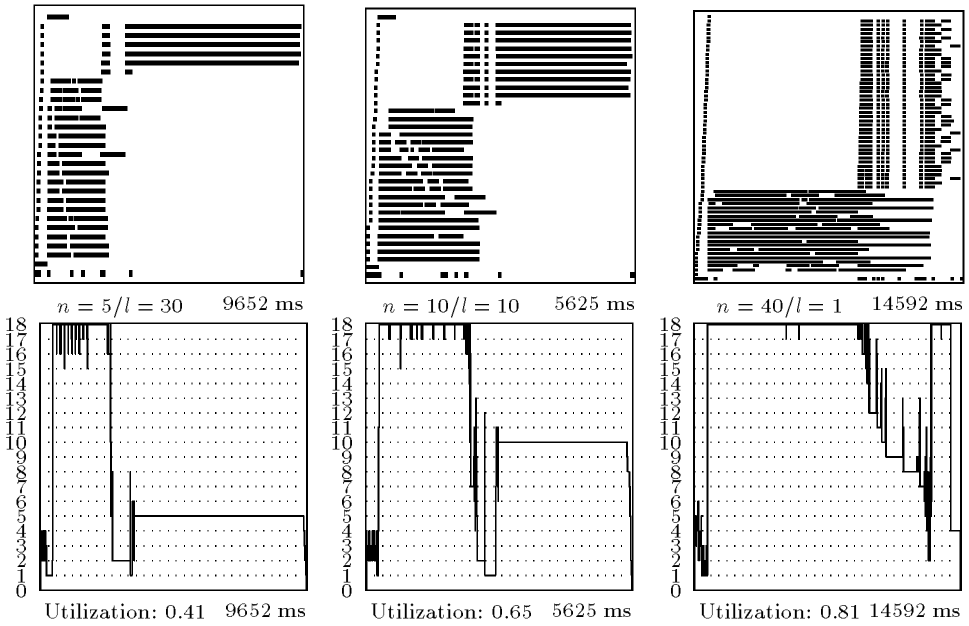

Figure 8: An Alternative Algorithm

to now. The algorithm therefore has to compute the inverse P_j^{-1} mod p_j where P_j is the product of the primes processed up to that moment.

There is an alternative formulation in which the result v is described as

$$v = \sum F(v^j, p_j, P) \bmod P$$

where P is the product of all primes p_j. Each summand can be independently (and concurrently) derived by a subformula F that takes as input arguments only the image v^j, the prime p_j and the product P and internally performs an inverse computation $(P/p_j)^{-1}$ mod p_j. Since the computation of this inverse is more time consuming than the computation of the inverse of the smaller P_j, the alternative formulation is usually not used in sequential implementations.

However, for a parallel implementation the new formulation may have some advantages: We can shift the computation of each summand totally into the corresponding `gauss` task; what then remains to do for each `crai` task is just to perform the summation and the final reduction y_i/d where d is computed by `crad`.

Figure 8 shows the outcome of this idea. The implementation differs from the previous versions in that the independent `gauss` tasks have been blocked into a fixed number of worker processes but this does not significantly change the runtime of the program. The important point is that the Chinese remaindering phase has become very short (the runtime of the `crad` task computing d can be neglected) and that almost all of the time in the `crai` tasks is now taken up by the sequential reduction y_i/d.

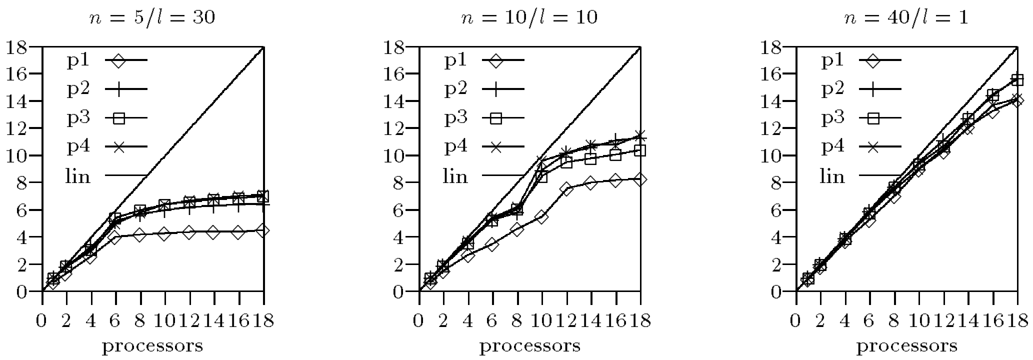

Figure 9: Speedup Comparisons

This optimization significantly decreases the runtime in the case of small and medium-sized equation systems. However, due to the larger total amount of work (and a worse scheduling behavior), it is slower in the case of the large equation system.

9 Experiments and Comparison

We finally present the speedups that could be achieved with the PACLIB implementation of the parallel algorithms compared to the sequential SACLIB implementation of the modular method. All speedups were computed on a Sequent Symmetry shared memory computer with 20 processors i80386 where 2 processors were not used in order to prevent system processes from interfering with the benchmarks. The results are displayed in Figure 9:

1. **n = 5/l = 30** : The "parallel cra" algorithm and the "alternative" algorithm performed best with a maximum speedup of more than 6. The "work shift" version was slightly worse while the "original" solution only yielded a speedup of 4.

2. **n = 10/l = 10** : The "alternative" version performed best with a speedup of almost 12 followed by the "work shift" version which even performed slightly better than the "parallel cra" version. The "original" solution was significantly worse with a speedup of 8.

3. **n = 40/l = 1** : The "work shift" version of the algorithm and the "parallel cra" version performed best with a speedup of almost 16. Both the "original" solution and the "alternative" solution were significantly worse.

We also investigated the influence of the dimension n and of the coefficient length l on the maximum speedup that could be achieved with 18 processors (see Figure 10):

1. **Dimension:** We fixed l to 3 and let n run from 5 to 40. For this relatively small coefficient length, no algorithm beats the "work shift" version for any n.

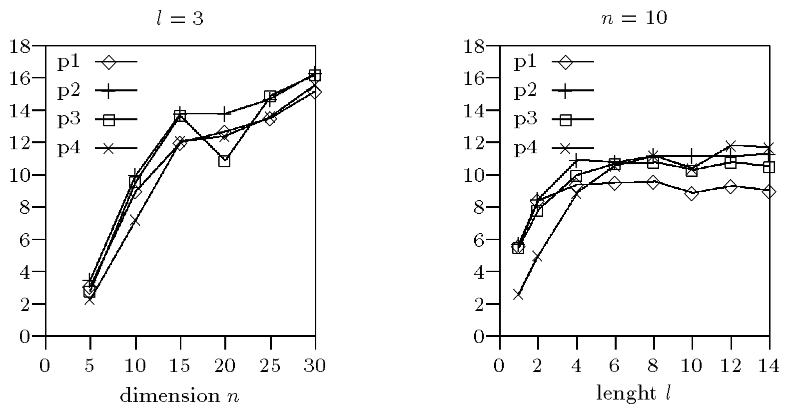

Figure 10: Influence of Dimension and Length

2. **Length:** We fixed n to 10 and let l run from 1 to 30. The "alternative" algorithm achieves with small l significantly lower speedups than the other algorithm but the more improves the larger l becomes and is best for $l > 12$

From these figures, it turns out that the simple "work shift" version is for large dimensions best and also looses for small dimensions not too much against the more optimized versions. However, if the dimension is small ($n \leq 10$) and the integer coefficients are very large ($l \geq 10$), the "alternative" formulation yields best performance and may be preferred.

10 Conclusions

We described several parallelization variants of the modular method for the exact solution of linear equation systems with large integer coefficients. If the dimension of the system is larger than the number of processors, we achieve very good speedups (more than 16 with 18 processors). By several improvements, we also manage for small systems with large coefficients to achieve speedups that are significantly larger than the dimension n of the system (a speedup of 7 for $n = 5$ and a speedup of 12 for $n = 10$). A main speedup limit for small systems is still the final reduction of the result. This involves the computation of the greatest common divisor for which no good parallel algorithm is known.

11 Acknowledgements

The authors thank Hoon Hong for initiating this work and for proof-reading this paper. Hans-Wolfgang Loidl's work on parallelizing the Chinese remaindering algorithm gave valuable inputs. This work was funded by the Austrian Research Agency FWF in the frame of the project S5302-PHY "Parallel Symbolic Computation".

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] B. Buchberger, G. E. Collins, R. Loos, and R. Albrecht, editors. *Computer Algebra — Symbolic and Algebraic Computation*. Springer, Vienna, New York, 1982.

[3] Peter A. Buhr and Richard A. Stroobosscher. The μSystem: Providing Light-weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software — Practice and Experience*, 20(9):929–964, September 1990.

[4] George Collins, Hoon Hong, Mark Encarnation, Jeremy Johnson, Werner Krandick, and Andreas Neubacher. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[5] George E. Forsythe and Cleve B. Moler. *Computer Solution of Linear Algebraic Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1967.

[6] V. G. Ganzha, E. V. Vorozhtsov, and J. A. Hulzen. A New Symbolic-Numeric Approach to Stability Analysis of Difference Schemes. In *ISSAC 92, International Symposium on Symbolic and Algebraic Computation*, Berkeley, California, July 27–29, 1992. ACM Press.

[7] C. M. Hoffman. Implicit Curves and Surfaces in Computer Aided Geometric Design. Technical Report CER-92-002, Department of Computer Science, Purdue University, 1992.

[8] Hoon Hong, Wolfgang Schreiner, Andreas Neubacher, Kurt Siegl, Hans-Wolfgang Loidl, Tudor Jebelean, and Peter Zettler. PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992. Also: Technical Report 92-9, ACPC Technical Report Series, Austrian Center for Parallel Computation, July 1992.

[9] Wolfgang Schreiner. The Design of the PACLIB Kernel. Technical Report 92-33, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[10] Volker Stahl. Solving a System of Linear Equations with Modular Arithmetic on a MIMD Computer. Technical Report 92-62, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[11] J. H. Wilkinson. The Evaluation of the Zeros of Ill-Conditioned Polynomials (Part I and II). *Numerische Mathematik*, 1:150–166,167–180, 1959.

[12] Franz Winkler. Computer Algebra I. Technical Report 88-88, RISC-Linz, Johannes Kepler University, Linz, Austria, 1988.